Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter

Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Leigh Orf

Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter

Matthieu Dorier, Gabriel Antoniu,
Franck Cappello, Marc Snir, Leigh Orf

ISSN 0249-6399   ISRN INRIA/RR--7706--FR+ENG

RESEARCH REPORT

N° 7706

November 2011

Project-Teams KerData
Joint INRIA/UIUC Laboratory
for Petascale Computing

Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter

Matthieu Dorier*, Gabriel Antoniu†,
Franck Cappello‡, Marc Snir§, Leigh Orf¶

Project-Teams KerData
Joint INRIA/UIUC Laboratory for Petascale Computing

Research Report n° 7706 — version 3 — initial version November 2011 — revised version April 2012 — 33 pages

* ENS Cachan Brittany, IRISA - Rennes, France. **matthieu.dorier@irisa.fr**
† INRIA Rennes Bretagne-Atlantique - France. **gabriel.antoniu@inria.fr**
‡ INRIA Saclay - France, University of Illinois at Urbana-Champaign - IL, USA. **fci@lri.fr**
§ University of Illinois at Urbana-Champaign - IL, USA. **snir@illinois.edu**
¶ Central Michigan University - MI, USA. **leigh.orf@cmich.edu**

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Abstract: With exascale computing on the horizon, the performance variability of I/O systems represents a key challenge in sustaining high performance. In many HPC applications, I/O is concurrently performed by all processes, which leads to I/O bursts. This causes resource contention and substantial variability of I/O performance, which significantly impacts the overall application performance together with the predictability of its run time. In this paper, we first illustrate the presence of I/O jitter on different platforms, and show the impact of different user-configurable parameters and I/O approaches on write performance variability. We then propose a new approach to I/O, called Damaris, which leverages dedicated I/O cores on each multicore SMP node, along with the use of shared-memory, to efficiently perform asynchronous data processing and I/O. We evaluate our approach on three different platforms including the Kraken Cray XT5 supercomputer, with the CM1 atmospheric model, which is one of the target HPC applications for the Blue Waters project. By overlapping I/O with computation and by gathering data into large files while avoiding synchronization between cores, our solution brings several benefits: 1) it fully hides the jitter as well as all I/O-related costs, which makes simulation performance predictable; 2) it increases the sustained write throughput by a factor of 15 compared to standard approaches; 3) it allows almost perfect scalability of the simulation where other I/O approaches fail to scale; 4) it enables a 600% compression ratio without any additional overhead, leading to a major reduction of storage requirements.

Key-words: Exascale Computing, Multicore Architectures, I/O, Variability, Dedicated Cores

Damaris: Exploitation du Parallelisme Multi-cœur pour Masquer la Variabilité des E/S

Résumé : Alors que l'ère de l'exascale se profile à l'horizon, la variabilité des performances des systèmes d'entrées-sorties (E/S) présente un défi majeur pour permettre d'atteindre des performances élevées et soutenues. Dans de nombreuses applications HPC, les E/S sont effectuées par tous les processus de manière concurrente. Cela engendre des piques d'utilisation des E/S, créant des conflits d'accès aux ressources et une variabilité substantielle dans les performances des E/S, impactant les performances globales de l'application de manière significative. Dans ce rapport, nous étudions l'influence de paramètres configurables par l'utilisateur sur la variabilitéé des E/S. Nous proposons une nouvelle approche, nommée Damaris, qui exploite des cœurs dédiés aux E/S sur chaque noeud SMP de manière à permettre des traitements et des mouvements de données efficaces et asynchrones. Nous évaluons notre approche sur trois plateformes, notamment sur le supercalculateur Kraken Cray XT5, avec l'application de modélisation atmosphérique CM1, une des applications visée par le projet Blue Waters. En rassemblant les données dans de plus gros fichiers tout en évitant la synchronisation entre les cœurs, notre solution apporte plusieurs avantages : 1) elle permit de cacher complètement la variabilité et les coûts liés aux E/S, permettant une prévisibilité des performances de l'application; 2) elle multiplie par 15 le débit d'écriture atteint en comparaison des approches standards; 3) elle permet un passage à l'échelle presque parfait de l'application là où les autres approaches échouent; 4) elle permet d'atteindre un taux de compression de 600% sans surcoût, induisant une réduction majeure de l'espace de stockage nécessaire.

Mots-clés : Exascale, Architectures Multi-cœurs, E/S, Variabilité, Cœurs Dédiés

Contents

1 Introduction

As HPC resources approaching millions of cores become a reality, science and engineering codes invariably must be modified in order to efficiently exploit these resources. A growing challenge in maintaining high performance is the presence of high variability in the effective throughput of codes performing input/output (I/O) operations. A typical I/O approach in large-scale simulations consists of alternating computation phases and I/O phases. Often due to explicit barriers or communication phases, all processes perform I/O at the same time, causing network and file system contention. It is commonly observed that some processes exploit a large fraction of the available bandwidth and quickly terminate their I/O, then remain idle (typically from several seconds to several minutes) waiting for slower processes to complete their I/O. This jitter has been observed at 1000-way scale, where measured I/O performance can vary by several orders of magnitude between the fastest and slowest processes [32]. This phenomenon is exacerbated by the fact that HPC resources are typically used many concurrent I/O intensive jobs. This creates file system contention between jobs, further increases the variability from one I/O phase to another and leads to unpredictable overall run times.

While most studies address I/O performance in terms of aggregate throughput and try to improve this metric by optimizing different levels of the I/O stack from the file system to the simulation-side I/O library, few efforts have been made in addressing I/O jitter. Yet it has been shown [32] that this variability is highly correlated with I/O performance, and that statistical studies can greatly help addressing some performance bottlenecks. The origins of this variability can substantially differ due to multiple factors, including the platform, the underlying file system, the network, and the application I/O pattern. For instance, using a single metadata server in the Lustre file system [11] causes a bottleneck when following the file-per-process approach (described in Section 2.2), a problem that PVFS [5] or GPFS [28] are less likely to exhibit. In contrast, byte-range locking in GPFS or equivalent mechanisms in Lustre cause lock contentions when writing to shared files. To address this issue, elaborate algorithms at the MPI-IO level are used in order to maintain a high throughput [26].

The contribution of this paper is twofold. We first perform an in-depth experimental evaluation of I/O jitter for various I/O-intensive patterns exhibited by the Interleaved Or Random (IOR) benchmark [29]. These experiments are carried out on the Grid'5000 testbed [2] with PVFS as the underlying file system, and on the Kraken Cray XT5 machine [22] at the National Institute for Computational Sciences (NICS), which uses Lustre. As opposed to traditional studies based on statistical metrics (average, standard deviation), we propose a new graphical method to study this variability and show the impact of some user-controllable parameters on this variability. Grid'5000 was used because it offers a root access to reserved nodes, and allows to deploy and tune our own operating system and file system. Thus we experiment on a clean environment by reserving entire clusters in order not to interfere with other users. In contrast, the Kraken larger-scale production supercomputer (currently 11^{th} in the Top500) is shared with other users.

Our second contribution aims to make one step further: It consists of an approach that hides the I/O jitter exhibited by most widely used approaches to I/O management in HPC simulations: the file-per-process and collective-I/O approaches (described in Section 2). Based on the observation that a first level of contention occurs when all cores of a multicore SMP node try to access the network for intensive I/O at the same time, our new approach to I/O, called Damaris (Dedicated Adaptable Middleware for Application Resources Inline Steering), leverages one core in each multicore SMP node along with shared memory to perform asynchronous data processing and I/O. These key design choices build on the observation that it is often not efficient to use all cores for computation, and that reserving one core for kernel tasks such as I/O management may not only help reducing jitter but also increase overall performance. Damaris takes into account user-provided information related to the application behavior and the intended use of the output in order to perform "smart" I/O and data processing within SMP nodes.

We evaluate the Damaris approach with the CM1 [3] application (one of the target applications for the Blue Waters [21] project) on three platforms, each featuring a different file system: Grid'5000 with PVFS, Kraken with Lustre and BluePrint, a Power5 cluster with GPFS. We compare Damaris to the classical file-per-process and collective-I/O approaches that have shown to greatly limit the scalability of the CM1 application and motivated our investigations. By using shared memory and by gathering data into large files while avoiding synchronization between cores, our solution achieves its main goal of fully hiding the I/O jitter. It thus allows CM1 to have a predictable run time and a perfect scalability. It also increases the I/O throughput by a factor of 6 compared to standard approaches on Grid'5000, and by a factor of 15 on Kraken, hides all I/O-related costs and enables a 600% compression ratio without any additional overhead.

This paper is organized as follows: Section 2 presents the background and motivations of our study. In Section 3 we run a set of benchmarks in order to characterize the correlation between the I/O jitter and a set of relevant parameters: the file striping policy, the I/O method employed (file-per-process or collective), the size of the data and the number of writers. We also perform a preliminary run of the CM1 application in order to illustrate the impact of the content of data on I/O variability. We experimentally illustrate the presence of jitter on Grid'5000 and on Kraken using a new graphical methodology. Damaris is presented in Section 4 together with an overview of its design and its API. We evaluate this approach with the CM1 atmospheric simulation running on 672 cores of the *parapluie* cluster on Grid'5000, 1024 cores of a Power5 cluster and on up to 9216 cores on Kraken. Section 6 finally presents a study of the theoretical and practical benefits of the approach, and positions it with respect to related works.

2 Background and related work

2.1 Understanding I/O jitter

Over the past several years, chip manufacturers have increasingly focused on multicore architectures, as the increase clock frequencies for individual processors has leveled off, primarily due to substantial increases in power consumption. These increasingly complex systems present new challenges to programmers, as approaches which work efficiently on simpler architectures often do not work well on these new systems. Specifically, high performance variability across individual components becomes more of an issue, and it can be very difficult to track the origin of performance weaknesses and bottlenecks. While most efforts today address performance issues and scalability for specific types of workloads and software or hardware components, few efforts are targeting the causes of performance variability. However, reducing this variability is critical, as it is an effective way to make more efficient use of these new computing platforms through improved predictability of the behavior and of the execution run time of applications. In [30], four causes of jitter are pointed out:

1. Resource contention within multicore SMP nodes, caused by several cores accessing shared caches, main memory and network devices.

2. Communication, causing synchronization between processes that run within a same node or on separate nodes. In particular, access contention for the network causes collective algorithms to suffer from variability in point-to-point communications.

3. Kernel process scheduling, together with the jitter introduced by the operating system.

4. Cross-application contention, which constitutes a random variability coming from simultaneous access to shared components in the computing platform.

While issues 3 and 4 cannot be addressed by the end-users of a platform, issues 1 and 2 can be better handled by tuning large-scale applications in such way that they make a more efficient use of resources. As an example, parallel file systems represent a well-known bottleneck and a source of high variability [32]. While the performance of computation phases of HPC applications is usually stable and only suffer from a small jitter due to the operating system, the time taken by a process to write some data can vary by several orders of magnitude from one process to another and from one iteration to another. In [15], variability is expressed in terms of *interferences*, with the distinction between *internal interferences* caused by access contention between the processes of an application (issue 2), and *external interferences* due to sharing the access to the file system with other applications, possibly running on different clusters (issue 4). As a consequence, adaptive I/O algorithms have been proposed [15] to allow a more efficient and less variable access to the file system.

2.2 Approaches to I/O management in HPC simulations

Two main approaches are typically used for performing I/O in large-scale HPC simulations:

The *file-per-process* approach it consists in having each process write in a separate, relatively small file. Whereas this avoids synchronization between processes, parallel file systems are not well suited for this type of load when scaling to hundreds of thousands of files: special optimizations are then necessary [4]. File systems using a single metadata server, such as Lustre, suffer from a bottleneck: simultaneous creations of so many files are serialized, which leads to immense I/O variability. Moreover, reading such a huge number of files for post-processing and visualization becomes intractable.

Using *collective I/O* e.g. in MPI applications, all processes synchronize together to open a shared file, and each process writes particular regions of this file. This approach requires a tight coupling between MPI and the underlying file system [26]. Algorithms termed as "two-phase I/O" [9, 31] enable efficient collective I/O implementations by aggregating requests and by adapting the write pattern to the file layout across multiple data servers [7]. When using a scientific data format such as pHDF5 [6] or pNetCDF [13], collective I/O avoids metadata redundancy as opposed to the file-per-process approach. However, it imposes additional process synchronization, leading to potential loss of efficiency in I/O operations. Moreover, pHDF5 and pNetCDF currently do not allow data compression.

It is usually possible to switch between these approaches when a scientific format is used on top of MPI; going from HDF5 to pHDF5 is a matter of adding a couple of lines of code, or simply changing the content of an XML file when using an adaptable layers such as ADIOS [16]. But users still have to find the best specific approach for their workload and choose the optimal parameters to achieve high performance and low variability. Moreover, the aforementioned approaches create periodic peak loads in the file system and suffer from contention at several levels. This first happens at the level of each multicore SMP node, as concurrent I/O requires all cores to access remote resources (networks, I/O servers) at the same time. Optimizations in collective-I/O implementations are provided to avoid this first level of contention; e.g. in ROMIO [8], data aggregation is performed to gather outputs from multiple cores to a subset of cores that interact with the file system by performing larger, contiguous writes.

3 Impact of user-controllable parameters on I/O variability

The influence of all relevant parameters involved in the I/O stack is not possible to ascertain in a single study: varying the network, the number of servers or clients, the file system together with its configuration, leads to too many degrees of freedom. Moreover, users can usually control only a restricted set of these parameters; we will therefore focus on such parameters only: the amount of data written at each I/O phase, the I/O approach used (collective write to a single shared file or individual files-per-process approach), the file striping policy inside the file system and the number of writing processes. The goal of this section is to emphasize their impact on I/O performance variability. We first present the the methodology and metrics we use to compare experiments, then we experimentally characterize I/O variability on the Grid'5000 French testbed and on Kraken using the IOR benchmark. Since I/O performance variability is a well known problem on Lustre (in particular due to its single metadata server [23]), we have used PVFS on Grid'5000 to show that I/O variability also appears on other systems and at smaller scale.

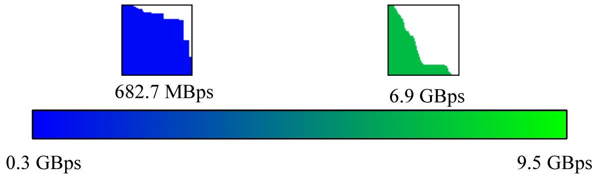


Figure 1: Visual representation of runs variability. The left run has a smaller throughput but less overall idle time than the right one.

3.1 Methodology

The main problem when studying variability instead of performance involves choosing the right metric and the proper way to quickly get valuable insight into the I/O systems behavior in a large number of experiments. Choosing the write variance as a statistical metric for the variability is arguably not appropriate for three reasons: first, it highly depends on the underlying configuration; second, the values are hard to interpret and compare; finally, providing decent confident intervals for such a statistic would require thousands of runs for each tested configuration. Thus, we need new methods to investigate I/O variability at large scale.

Visual representation of runs

We propose the following methodology to study the variability of I/O systems. We define a *run* as a single I/O phase of an application under a particular configuration. An *experiment* is a set of runs with the same configuration. For a particular run, we measure the time taken by each process to write its data. This time is reported on a small graph (that we call a *tile*), in which processes are sorted vertically by their write time, this time being represented by an horizontal line. We call the *variability* pattern the shape within a particular tile. As will be shown, for a given configuration we find recurrent patterns. And example of such a representation is shown in Figure 1

The throughput of a run, defined as the total amount of data divided by the time of the slowest process, is shown using a color scale for drawing these graphs, and also printed under each tile[1]. Compared to a computation of variance only, this method presents a more complete picture, enabling us to concurrently visualize different types of jitter including variability from a process to another, from a run to another and between experiments. The ratio between the white part and the colored part of a tile gives an overview of the time wasted by processes waiting for the slowest to complete the write. The shape within tiles provides some clues about the nature of the variability. The range of colors within a set of tiles provides a visual indication of the performance variability of a configuration for a given experiment, and also lets us compare experiments when the same color scale is used. In the following discussion, only a subset of representative runs are presented for each experiment.

[1] Note: this representation is color-based, black and white printing may lead to a loss of information.

Platforms and tools considered

The first experiments have been conducted on two clusters in the Rennes site of the French Grid'5000. The *parapluie* cluster, featuring 40 nodes (2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM), is used for running the IOR benchmark on a total of 576 cores (24 nodes). The latest version OrangeFS 2.8.3 of PVFS is deployed on 16 nodes of the *parapide* cluster (2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk), each node used both as I/O server and metadata server. In each cluster all nodes are communicate through a 20G InfiniBand 4x QDR link connected to a common Voltaire switch. We use MPICH [20] with ROMIO [31] compiled against the PVFS library.

The last experiments aim to show the evolution of the jitter at large process counts. They are conducted on Kraken, the Cray XT5 machine running at NICS. Kraken has 9408 compute nodes running Linux Cray. Each node features 12 cores (two 2.6 GHz six-core AMD Opteron processors) and 16 GB of main memory, nodes are connected to each other through a Cray SeaStar2+ router. The Lustre file system available on Kraken is deployed on 336 OST (Object Storage Targets) running on 48 OSS (Object Storage Servers), and one MDS (MetaData Server). Kraken is shared by all users of the platforms as we are running experiments.

IOR [29] is a benchmark used to evaluate I/O optimizations in high performance I/O libraries and file systems. It provides backends for MPI-IO, POSIX and HDF5, and lets the user configure the amount of data written per client, the transfer size, the pattern within files, together with other parameters. We modified IOR in such a way that each process outputs the time spent writing; thus we have a process-level trace instead of simple average of aggregate throughput and standard deviation.

3.2 Results with the IOR benchmark

3.2.1 Impact of the data size

We first study the impact of the size of the data written by the processes on overall performance and performance variability. We perform a set of 5 experiments on Grid'5000 with a data size ranging from 4 MB to 64 MB per process using 576 writers. These data are written in a single shared file using collective I/O. Figure 2 represents 5 of these runs for each data size. We notice that for small sizes (4 MB) the behavior is that of a waiting queue, with a nearly uniform probability for a process to write in a given amount of time. However, the throughput is highly variable from one run to another, ranging from 4 to 8 GB/s. An interesting pattern is found when each process outputs 16 MB: the idle time decreases, the overall throughput is higher (around 7 GB/s) and more regular from a run to another. Moreover the variability pattern shows a more fair sharing of resources. Performance then decreases with 32 and 64 MB per process. When writing 64 MB per process, the throughput is almost divided by 7 compared to writing 16 MB.

IOR allows the user to configure the transfer size, i.e., the number of atomic write calls in a single phase. When this transfer size is set to 64 KB (which is equal to the MTU of our network and to the stripe size in the file system), we observed that writing 4, 8 or 16 MB per process leads to the same variability pattern and throughput (around 4 GB/s). Increasing the total data size to 32 or

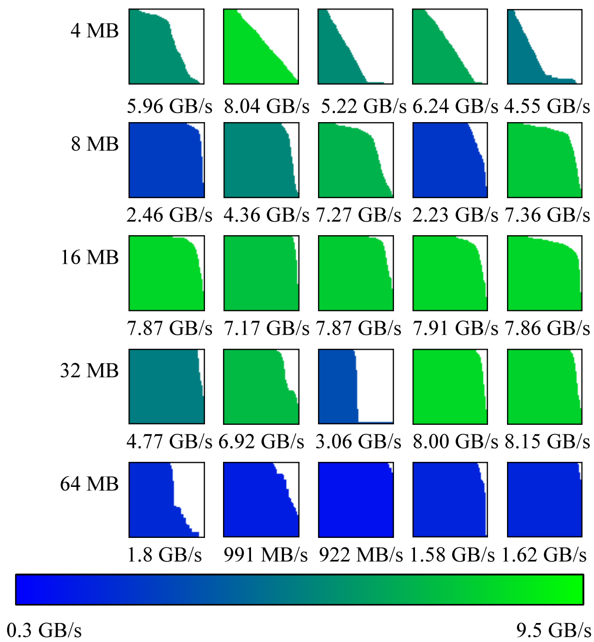


Figure 2: Impact of the data size on I/O performance variability. Experiments are conducted on 576 cores of Grid'5000, using Collective I/O.

64 MB shows however a decrease of performance and an increase of variability from one run to another. With 64 MB, a staircase pattern starts to appear, which might be due to the amount of data reaching cache capacity limits in the file system.

3.2.2 Impact of the striping policy

Like Lustre, PVFS allows the user to tune the file's striping policy across the I/O servers using `lfs setstripe` commands. The default distribution in PVFS consists in striping data in a round robin manner across the different I/O servers. The stripe size is one of the parameters that can be tuned. The PVFS developers informed us that given the optimizations included in PVFS and in MPICH, changing the default stripe size (64 KB) should not induce any change in aggregate throughput. We ran IOR three times, making each of the 576 processes writes 32 MB in a single write within a shared file using collective I/O. We observed that a similar maximum throughput of around 8 GB/s is achieved for both 64 KB, 4 MB and 32 MB for stripe size. However, the throughput was lower on average with a stripe size of 4 MB and 32 MB, and the variability patterns, which can be seen in Figure 3, shows more time wasted by processes waiting.

While the stripe size had little influence on collective I/O performance, this was not the case for the file-per-process approach. When writing more than 16 MB per process, the lack of synchronization between processes causes a huge access contention and some of the write operations to time out. The generally-accepted approach of increasing the stripe size or reducing the stripes count for

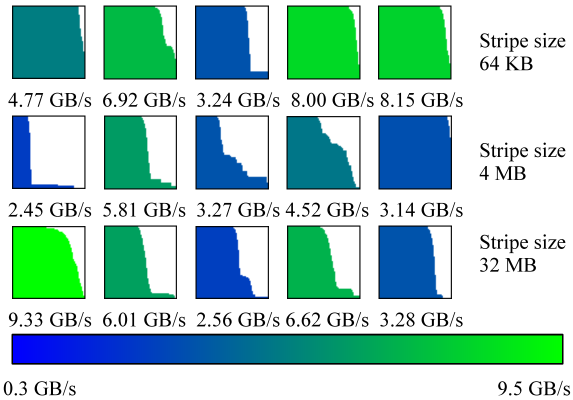


Figure 3: Throughput variability for three experiments with different stripe size. Experiments conducted on 576 cores of Grid'5000, using Collective I/O.

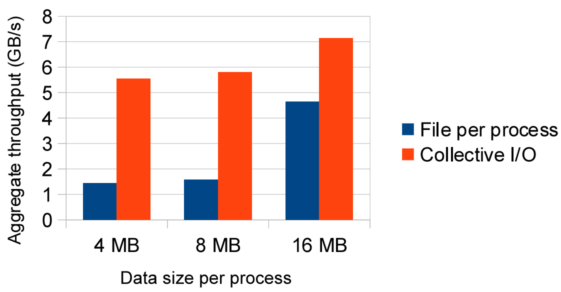


Figure 4: Aggregate throughput comparison between file-per-process and Collective I/O on 576 processes of Grid'5000.

the many-file approach is thus justified, but as we will see, this approach still suffers from high variability.

3.2.3 Impact of the I/O approach

We then compared the collective I/O approach with the file-per-process approach. 576 processes are again writing concurrently in PVFS. With a file size of 4 MB, 8 MB and 16 MB and a stripe size of 64 KB in PVFS, the file-per-process approach shows a much lower aggregate throughput than collective I/O. Results (in terms of average aggregate throughput) are shown on Figure 4. Figure 5 presents the recurrent variability patterns of runs for 8 MB per process (runs writing 4 MB and 16 MB have similar patterns). We observe a staircase progression, which suggests communications timing out and processes retrying in series of bursts.

As previously mentioned, writing more data per client required to change the stripe size in PVFS. Yet the throughput obtained when writing 32 MB per process using a 32 MB stripe size and the file-per-process approach on Grid'5000

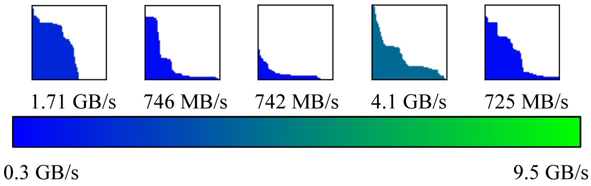


Figure 5: Throughput variability with the file-per-process approach on 576 cores of Grid'5000, each core writing 32 MB.

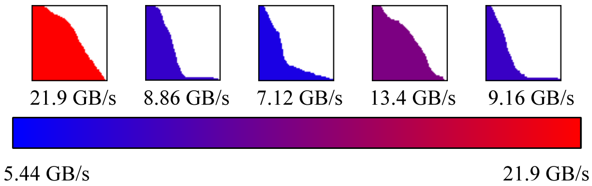


Figure 6: Throughput variability with the file-per-process approach on Grid'5000, writing 32 MB per process with a 32 MB stripe size.

is found to be the highest over all experiments we did, with a peak throughput of more than 21 GB/sec. Thus Figure 6 shows the results with a different color scale in order not to be confused with other experiments. We notice that despite the high average and peak throughput, there are periods where a large number of processes are idle, and there is a high variance from a run to another.

3.2.4 Impact of the number of writers

On Kraken, we reproduced the experiment presented in Figure 5 of [23]: each process outputs 128 MB using a 32 MB transfer size and a file-per-process approach. Results are presented in Figure 7: the aggregate throughput increases with the number of processes, just as expected in [23]. An intuitive idea could have been that with larger process counts comes a higher variability from a run to another. This is however not the case. The explanation behind this is the law of large numbers: as the number of processes increases and since each process has to issue several transfers, we are more likely to observe a gaussian repartition of the time spent by a process writing its data. Indeed, with 8400 writers the variability patterns start to look like the cumulative distribution function of a gaussian. This effect has already been exemplified in [32].

3.3 Preliminary results with CM1

We run a preliminary experiment on Kraken to illustrate the presence of I/O jitter in the CM1 application, which will be further presented in Section 5. CM1 outputs one file per process using the HDF5 format. Considering the I/O tips provided on the Kraken website [23], we set the file system's parameters so that each file (approximately 26 MB) is handled by a single storage server. Figure 8 represents the variability patterns of five write phases using three

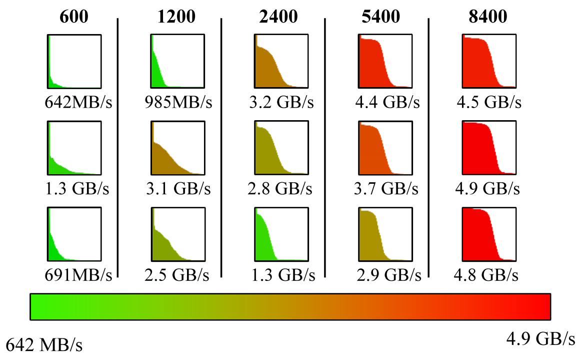


Figure 7: Throughput variability with the file-per-process approach on Kraken, with different number of writers. Each process writes 128 MB with a 32 MB transfer size.

different configurations: 576 or 2304 core, compression enabled[2] or not.

As we can see, a lot of time is wasted waiting for slow processes with the file-per-process approach, both when using 576 and 2304 cores. The variability from a write phase to another is also clear. When enabling compression, the write time depends on the duration of the compression phase and he size of the compressed data, thus adding more variability.

In conclusion, the experiments presented above show that the performance of I/O phases is extremely variable, and that this variability, together with the average and peak throughput achieved, highly depends on the data size, on the I/O approach and the number of writers. The content of the data itself can have a big impact when high-level data formats perform indexing or compression before writing, leading to higher performance unpredictability. Experiments conducted on Grid'5000 with clusters fully reserved to our own experiments (including network interconnects) prove that a high variability can already be experienced with few concurrent accesses to the parallel file system. Whereas tuning the stripe size in PVFS has almost no influence when using collective I/O, it greatly helps to achieve a very high throughput with the file-per-process approach. We have successfully demonstrated that the chosen parameters and the I/O approach have a big impact on the performance and variability of I/O phases. The last experiment on Grid'5000 revealed that while a high throughput of at least 21 GB/s can be expected of our file system, poor configuration choices can lead to barely a few hundreds of MB/s.

The experiments conducted on Kraken with respect to scalability showed that in the presence of many independent writes, the variability pattern is becoming more predictable as we increase the number of writers. Still, we notice that a small set of processes manage to quickly terminate their I/O phase, which motivates further research of new I/O approaches that do not exhibit this variability.

[2]The represented throughput corresponds to the apparent aggregate throughput, i.e. the total amount of raw uncompressed data divided by the time for the slowest process to complete the write phase.

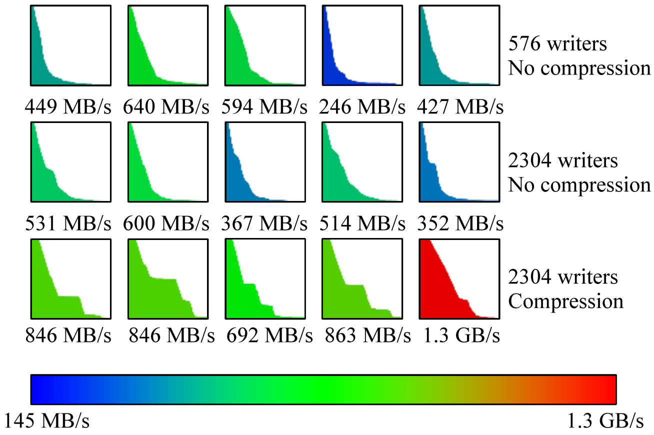


Figure 8: Variability patterns for three I/O phases of CM1 on Kraken, using different numbers of cores and enabling/disabling compression. A high variability is shown in both configurations, either from a process to another and from an iteration to another.

4 The dedicated core approach

One finding of our study of I/O performance variability is that, in order to sustain a high throughput and a lower variability, it is preferable to avoid as much as possible access contentions at the level of the network interface and of the file system, for example by reducing the number of writers and the number of generated files. As the first level of contention occurs when several cores in a single SMP node try to access the same network interface, it becomes natural to work at the level of a node.

We propose to gather the I/O operations into one single core that will perform writes of larger data in each SMP node. In addition, this core is dedicated to I/O (i.e. will not run the simulation code) in order to overlap writes with computation and avoid contention for accesses to the file system. The communication between cores running the simulation and dedicated cores is done through shared-memory in order to make a write as fast as a `memcpy`. We call this approach Damaris (Dedicated Adaptable Middleware for Application Resources Inline Steering): this section introduces its design, implementation and API.

4.1 Principle

Damaris consists of a set of servers, each of which runs on a dedicated core in every SMP node used by the simulation. This server keeps data in a shared memory segment and performs post-processing, filtering and indexing in response to user-defined events sent either by the simulation or by external tools.

The buffering system running on these dedicated cores includes metadata information about incoming variables. In other words, clients do not write raw data but enriched datasets in a way similar to scientific data formats such as HDF5 or NetCDF. Thus dedicated cores have full knowledge of incoming datasets and can perform “smart actions” on these data, such as writing important datasets in priority, performing compression, statistical studies, indexing, or any user-provided transformation. By analyzing the data, the notion of an “important dataset” can be based on the scientific content of the data and thus dynamically computed, a task that low-level I/O scheduler could not perform.

4.2 Architecture

Figure 9 presents the architecture of Damaris, by representing a multicore SMP node in which one core is dedicated to Damaris. The other cores (only three represented here) are used by the simulation. As the number of cores per node increases, dedicating one core has a diminishing impact. Thus, our approach primarily targets SMP nodes featuring a large number of cores per node (12 to 24 in our experiments).

Communication between the computation cores and the dedicated cores is done through shared memory. A large memory buffer is created by the dedicated core at start time, with a size chosen by the user. Thus the user has a full control over the resources allocated to Damaris. When a compute core submits new data, it reserves a segment of this buffer, then copies its data using the returned pointer. Damaris uses the default mutex-based allocation algorithm of the Boost library to allow concurrent atomic allocations of segments by multiple clients. We also implemented another lock-free allocation algorithm: when all clients are expected to write the same amount of data, the buffer is split in as many parts as clients and each client uses its own region to communicate with Damaris.

To avoid using the shared memory to transfer too much metadata information, we follow a design concept proposed by ADIOS [16] of using an external configuration file to provide static information about the data (such as names, description, unit, dimensions...). The goals of this external configuration are 1) to keep a high-level description of the datasets within the server, allowing higher-level data manipulations, 2) to avoid static layout descriptions to be sent by clients through the shared memory (only data is sent together with the minimal descriptor that lets the server retrieve the full description in its metadata system).

The event-queue is another shared component of the Damaris architecture. It is used by clients either to inform the server that a write completed (*write-notification*), or to send *user-defined events*. The messages are pulled by an event processing engine (EPE) on the server side. The configuration file also includes information about the actions to perform upon reception of an event. Such actions can prepare data for future analysis, or simply write it using any I/O library.

All chunks of data written by the clients are characterized by a tuple ⟨*name, iteration, source, layout*⟩. *Iteration* gives the current step of the simulation, while *source* uniquely characterizes the sender (e.g. its MPI rank). The *layout* is a description of the structure of the data: type, number of dimensions and extents. For most simulations, this layout does not vary at runtime and can be provided also by the configuration file. Upon reception of a write-notification, the EPE will add an entry in a metadata structure associating the tuple with the received data. The data stay in shared memory until actions are performed on them.

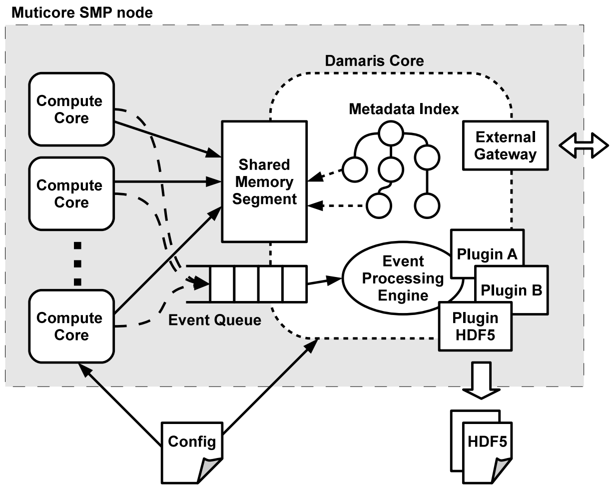


Figure 9: Design of the Damaris approach.

4.3 Key design choices

Behavior management and user-defined actions

The EPE can be enriched by plugins provided by the user. A plugin is a function (or a set of functions) embedded in a dynamic library or in a Python script that the actions manager will load and call in response to events sent by the application. The matching between events and expected reactions is provided by the external configuration file. Thus, it is easy for the user to define a precise behavior for Damaris by simply changing the configuration file. Our intents when designing Damaris was to provide a very simple way for users to extend it and adapt it to the particular needs of their simulations.

Minimum-copy overhead

The efficiency of interactions between clients and dedicated cores is another strength of Damaris. At most a single copy from a local variable to the shared memory buffer is required to send data to the dedicated core. Damaris also includes the possibility to "write" data without actually making a copy: the simulation directly allocates its variables in the shared memory buffer. When the simulation finishes working on an array, it simply informs the dedicated core that the data can be considered as ready. This is a strong point of Damaris that distinguishes it from other dedicated-process-based approaches [17, 14], in which multiple raw copies of the data have to be moved using either the network or kernel functionalities (such as a FUSE interface).

Event scope

It can be desirable to force a reaction only when a given subset of computation cores have sent an event. This can be done using the notion of event scope, which defines the subset of processes that has to perform a synchronized action, such as statistical data sharing for establishing a global diagnosis of the simulation. This synchronization involves distributed algorithms such as total order broadcast, and efficient scheduling schemes, that are beyond the scope of this paper.

Interaction semantics

The use of a single message queue shared by all the cores and used both for user-defined events and write notifications ensures that a sequence of events or data sent by a compute core will be treated in the same order in the dedicated core. Yet, sequences of events coming from different compute cores may interleave. This semantics is important because the purpose of user-defined events is to force a behavior at a given execution point of the client, knowing what has already been written.

Persistency layer

Damaris interfaces with an I/O library such as HDF5 by using a custom persistency layer with HPF5 callback routines embedded in a plugin, as shown as an example in Figure 9.

4.4 Client-side API

Damaris is intended to be a generic, platform-independent, application-independent, easy-to-use tool. The current implementation is developed in C++ and uses the Boost library for interprocess communications and options parsing, and Xerces-C for XML configuration. It provides client-side interfaces for C, C++ and Fortran applications and requires only few minor changes in the application's I/O routines, together with an external configuration file that describes the data. The client-side interface is extremely simple and consists in four main functions (here in C):

- `dc_initialize("config.xml",core_id)` initializes the client by providing a configuration file and a client ID (usually the MPI rank). The configuration file will be used to retrieve knowledge about the expected data layouts.
- `dc_write("varname",step,data)` pushes some data into the Damaris' shared buffer and sends a message to Damaris notifying the incoming data. The dataset being sent is characterized by its name and the corresponding iteration of the application. All additional information such as the size of the data its layout (including its datatype) and additional descriptions are provided by the configuration file.
- `dc_signal("eventname",step)` sends a custom event to the server in order to force a behavior predefined in the configuration file.
- `dc_finalize()` frees all resources associated with the client. The server is also notified of client's disconnection and will not accept any other incoming data or event from this client.

Additional functions are available to allow direct access to an allocated portion of the shared buffer (`dc_alloc` and `dc_commit`), avoiding an extra copy from local memory to shared memory. Other functions allow the user to dynamically modify the internal configuration initially provided, which can be useful when writing variable-length arrays (which is the case in particle-based simulations, for example) or user-defined datatypes.

Below is an example of a Fortran program that makes use of Damaris to write a 3D array then send an event to the I/O core. The associated configuration file, which follows, describes the data that is expected to be received by the I/O core, and the action to perform upon reception of the event.

```
program example
  integer :: ierr, rank, step
  real, dimension(64,16,2) :: my_data
  call df_initialize("my_config.xml", rank, ierr)
  ...
  call df_write("my_variable", step, my_data, ierr)
  call df_signal("my_event", step, ierr)
  ...
  call df_finalize(ierr)
end program example
```

Associated XML configuration file:

```
<layout name="my_layout" type="real"
        dimensions="64,16,2" language="fortran" />
<variable name="my_variable" layout="my_layout" />
<event name="my_event" action="do_something"
        using="my_plugin.so" scope="local" />
```

5 Experimental results with CM1

We have evaluated our approach based on dedicated I/O cores against standard approaches with the CM1 atmospheric simulation, using three platforms: Grid'5000 and Kraken (already presented before), and BluePrint, a Power5 cluster.

5.1 The CM1 application

CM1 [3] is used for atmospheric research and is suitable for modeling small-scale atmosphere phenomena such as thunderstorms and tornadoes. It follows a typical behavior of scientific simulations which alternate computation phases and I/O phases. The simulated domain is a fixed 3D array representing part of the atmosphere. The number of points along the x, y and z axes is given by the parameters n_x, n_y and n_z. Each point in this domain is characterized by a set of variables such as local *temperature* or *wind speed*.

CM1 is written in Fortran 95. Parallelization is done using MPI, by splitting the 3D array along a 2D grid. Each MPI rank is mapped into a pair (i, j) and simulates a $n_{sx} * n_{sy} * n_z$ points subdomain. In the current release (r15), the I/O phase uses HDF5 to write one file per process. Alternatively, the latest development version allows the use of pHDF5. One of the advantages of using a file-per-process approach is that compression can be enabled, which is not the case with pHDF5. Using lossless gzip compression on the 3D arrays, we observed a compression ratio of 187%. When writing data for offline visualization, the floating point precision can also be reduced to 16 bits, leading to nearly

600% compression ratio when coupling with gzip. Therefore, compression can significantly reduce I/O time and storage space. However, enabling compression leads to an additional overhead, together with more variability both from a process to another, and from a write phase to another.

5.2 Platforms and configuration

- **On Kraken** we study the scalability of different approaches, including Damaris. Thus, the problem size vary from an experiment to another. When all cores are used by the simulation, each process handles a 44x44x200 points subdomain. Using Damaris, each non-dedicated core (11 out of 12) handles a 48x44x200 points subdomain, thus making the total problem size equivalent.

- **On Grid'5000** we run the development version of CM1 on 28 nodes, i.e. 672 cores. The total domain size is 1104x1120x200 points, so each core handles a 46x40x200 points subdomain with a standard approach, and a 48x40x200 points subdomain when one core out of 24 is used by Damaris.

- **BluePrint** provides 120 Power5 nodes. Each node consists in 16 cores and features 64 GB of memory. The GPFS file system is deployed on 2 dedicated nodes. CM1 was run on 64 nodes (1024 cores), with a $960\times960\times300$ points domain. Each core handled a $30 \times 30 \times 300$ points subdomain with the standard approach. When dedicating one core out of 16 on each node, computation cores handled a $24\times40\times300$ points subdomain. On this platform we vary the size of the output by enabling or disabling variables. We enabled the compression feature of HDF5 for all the experiments done on this platform.

CM1 takes an input file indicating whether each variable should be written or not, together with the frequency of outputs. We ran it with an output configuration and frequency corresponding to what can be expected by an atmospheric scientist from such a simulation.

5.3 Experimental results

In this section, we present the results achieved in terms of I/O jitter, I/O performance and resulting scalability of the application.

5.3.1 Effect on I/O jitter

Figure 10 shows the average and maximum duration of an I/O phase on Kraken from the point of view of the simulation. It corresponds to the time between the two barriers delimiting the I/O phase. As we can see, this time is extremely high and variable with Collective-I/O, achieving up to 800 sec on 9216 processes. The average of 481 sec represents about 70% of the overall simulation's run time, which is simply unacceptable. By setting the stripe size to 32 MB instead of 1 MB in Lustre, the write time went up to 1600 sec with Collective-I/O, exemplifying the fact that bad choices of file system's configuration can lead to extremely poor I/O performance. Unexpectedly, the file-per-process approach seemed to lead to a lower variability, especially at large process count. Yet it still represents an unpredictability (difference between the fastest and the slowest phase) of about ±17 *sec*. For a one month run, writing every 2 minutes would lead to an uncertainty of several hours to several days of run time. When using Damaris, we dedicate one core out of 12 on each SMP node, thus potentially reducing the computation power for the benefit of I/O efficiency (the impact on overall application performance is discussed in the next section). As a means to reduce I/O jitter, this approach is clearly effective: the time to write from the point of view of the simulation is cut down to the time required to perform a series of copies in shared memory. It leads to a write time of 0.2 sec and does not depend anymore on the number of processes. The variability is in order of 0.1 sec (which cannot be represented in this figure).

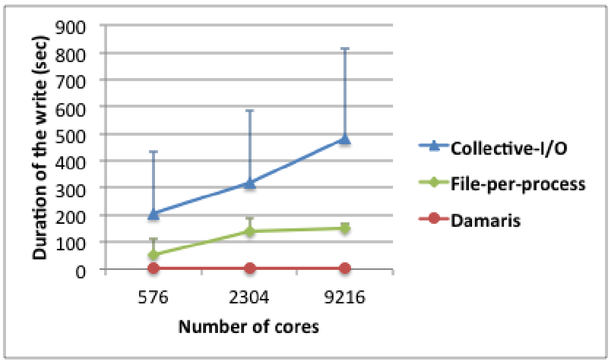


Figure 10: duration of a write phase on Kraken (average and maximum). For readability reasons we don't plot the minimum write time. Damaris shows to completely remove the I/O jitter while file-per-process and collective-I/O have a big impact on the runtime predictability.

We also have evaluated the behavior the CM1's file-per-process method on BluePrint with different output configurations, ranging from a couple of 2D arrays to a full backup of all 3D arrays. Compression was enabled, but the amount of data will be reported as logical data instead of physical bytes stored. By mapping contiguous subdomains to cores located in the same node, we have also been able to implement a filter that aggregates datasets prior to actual I/O. The number of files is divided by 16, leading to 64 files created per I/O phase instead of 1024. With a dedicated core, CM1 reports spending less than 0.1 sec in the I/O phase, that is to say only the time required to perform a copy of the data in the shared buffer. The results are presented in Figure 11. As we increase the amount of data, we increase the variability of the I/O time with the file-per-process approach. With Damaris however, the write time remains in the order of 0.2 sec for the largest amount of data and the variability in the order of 0.1 sec again.

Similar experiments have been carried out on Grid5000. We ran CM1 using 672 cores, writing a total of 15.8 GB uncompressed data (about 24 MB per process) every 20 iterations. With the file-per-process approach, CM1 reported spending 4.22% of its time in I/O phases. Yet the fastest processes usually terminate their I/O in less than 1 sec, while the slowest take more than 25 sec. Figure 12 (a) shows the CPU usage with a file-per-process approach on a par-

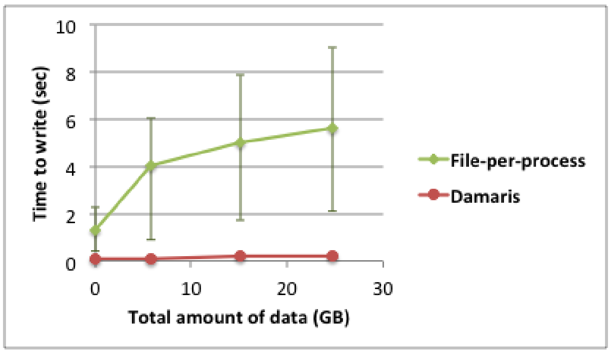
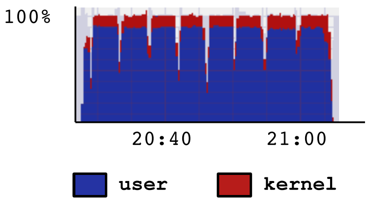
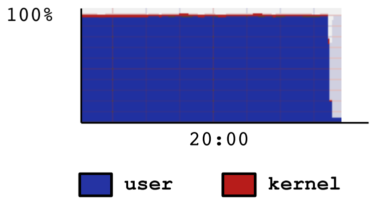


Figure 11: duration of a write phase (average, maximum and minimum) using file-per-process and Damaris on BluePrint (1024 cores). The amount of data is given in total per write phase.



(a) CPU usage with Collective I/O



(b) CPU usage with Damaris

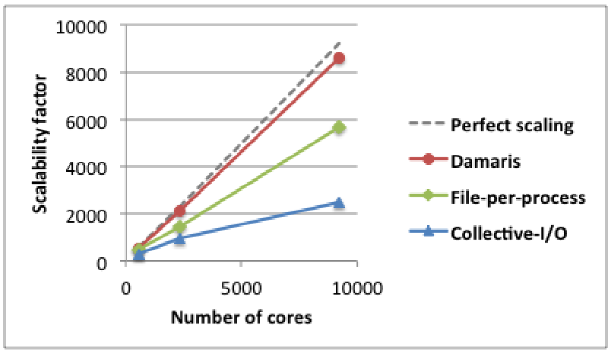
Figure 12: CPU usage measured on a node running CM1 with a collective I/O mechanism (a) and with Damaris (b).

ticular node during a run of CM1. Every I/O phase breaks the computation and lower the resource usage. Damaris on the other hand is fully capable of utilizing the resources available on a node, as shown in Figure 12 (b).

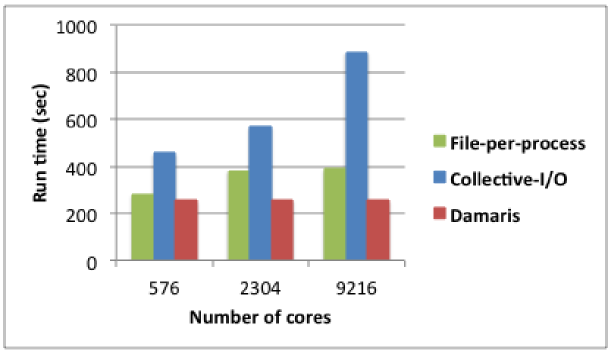
These experiments show that by replacing write phases by simple copies in shared memory and by leaving the task of performing actual I/O to a dedicated core, Damaris is able to completely hide the I/O jitter from the point of view of the simulation, making the application runtime more predictable and allowing a better use of de node's resources.

5.3.2 Application's scalability and I/O overlap

CM1 exhibits very good weak scalability and very stable performance when no I/O is performed. Thus as we increase the number of cores, the scalability becomes mainly driven by the I/O phase. To measure the scalability of an approach, we define the scalability factor S of an approach as $S = N * \frac{C_{576}}{T_N}$ where N is the number of cores considered. We take as a baseline the time C_{576} of 50 iterations of CM1 on 576 processes without dedicated core and without any I/O. T_N is the time CM1 takes to perform 50 iterations plus one I/O phase, on N cores. A perfect scalability factor on N cores should be N. The scalability factor on Kraken for the three approaches is given in Figure 13 (a). Figure 13



(a) Scalability factor on Kraken



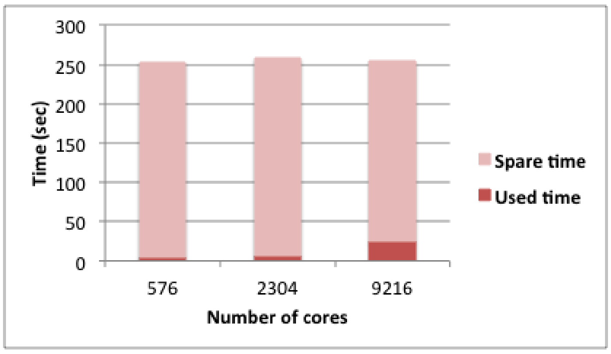
(b) Run time on Kraken

Figure 13: (a) Scalability factor and (b) overall run time of the CM1 simulation for 50 iterations and 1 write phase, on Kraken.

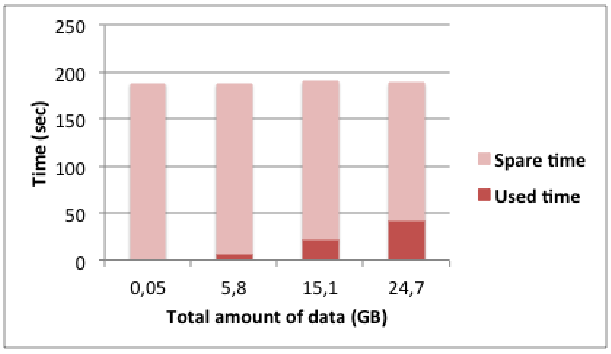
(b) provides the associated application run time for 50 iterations plus one write phase. As we can see, Damaris shows a nearly perfect scalability where other approaches fail to scale. The difference between Damaris' scalability and the perfect scalability comes from two facts: 1) CM1 itself shows good but not perfect scalability and 2) as some cores are dedicated to I/O, the computation time for the same payload may have increased.

Since the scalability of our approach comes from the fact that I/O overlap with computation, we have to show that the dedicated cores have enough time to perform the actual I/O while computation goes on. Figure 14 shows the time used by the dedicated cores to perform the I/O on Kraken and BluePrint, as well as the time they spare for other usage. As the amount of data on each node is equal, the only explanation for the dedicated cores to take more time at larger process count on Kraken is the access contention for the network and the file system. On BluePrint the number of processes here is constant for each experiments, thus the differences in the times to write come from the different amounts of data.

Similar results (not presented because of space constraints) have been achieved on Grid'5000. On all platforms, Damaris shows that it can fully overlap writes with computation and still remain idle 75% to 99% of time. Thus without im-



(a) Write time / Spare time on Kraken



(b) Write time / Spare time on BluePrint

Figure 14: Time spent by the dedicated cores writing data for each iteration, and time spared. The spare time is the time dedicated cores are not performing any task.

pacting the application, we could further increase the frequency of outputs or perform inline data analysis, as mentioned in Section 4. This second approach will be subject to a future paper.

5.3.3 Effective I/O performance

Figure 15 presents the aggregate throughput obtained with the different approaches on Kraken. Note that in the case of Damaris, this throughput is only seen by the dedicated cores. Damaris achieves an aggregate throughput about 6 times higher than the file-per-process approach, and 15 times higher than Collective I/O. This is due to the fact that Damaris avoids process synchronization and access contentions at the level of a node. Also by gathering data into bigger files, it reduces the pressure on metadata servers and issues bigger operations that can be more efficiently handled by storage servers.

The results obtained on 672 cores of Grid'5000 are presented in Table 1. The throughput achieved with Damaris is more than 6 times higher than standard approaches.

Since compression was enabled on BluePrint, we don't provide the resulting throughputs, as it would depend on the overhead of the compression algorithm

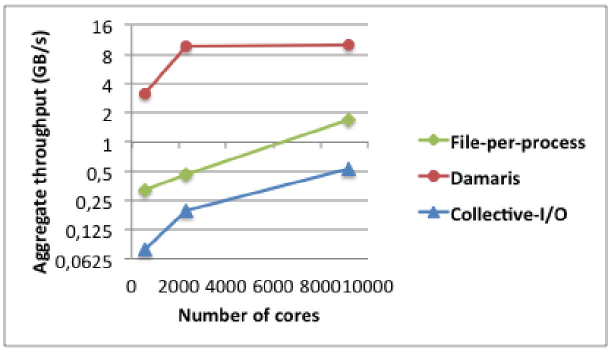


Figure 15: Average aggregate throughput achieved on Kraken with the different approaches.

	Average aggregate throughput
File-per-process	695 MB/s
Collective-I/O	636 MB/s
Damaris	4.32 GB/s

Table 1: Average aggregate throughput on Grid'5000, with CM1 running on 672 cores.

and the size of the data.

A potential improvement in the file-per-process approach can be found by buffering several outputs using the `H5P_set_faple_core` feature of HDF5, which allows the datasets to be written in local memory an flushed only when closing the file. The study of this method is left as future work.

In conclusion, reducing the number of writers while gathering data into bigger files also has an impact on the throughput that the simulation can achieve. On every platform, Damaris substantially increases throughput, thus making a more efficient use of the file system.

5.4 Potential use of spare time

In order to leverage spare time in the dedicated cores, we implemented several improvements.

5.4.1 Compression

As mentioned before, using lossless gzip compression on the 3D arrays, we observed a compression ratio of 187%. When writing data for offline visualization, the floating point precision can also be reduced to 16 bits, leading to nearly 600% compression ratio when coupling with gzip. We achieved an apparent throughput of 4.1 GB/s from the point of view of the dedicated cores. Contrary to enabling compression in the file-per-process approach (which has been exemplified in Section 3), the overhead and jitter induced by this compression is completely hidden within the dedicated cores, and do not impact the running simulation. In addition, by aggregating the data into bigger files, we reduce its

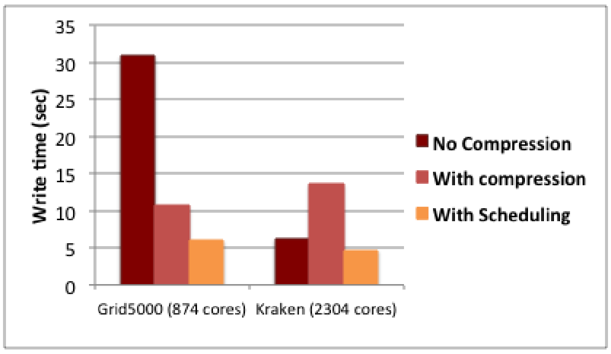


Figure 16: Write time in the dedicated cores when enabling the compression feature and the scheduling strategy.

total entropy, thus increasing the achievable compression ratio.

5.4.2 Data transfer scheduling

Additionally, we implemented in Damaris the capability to schedule data movements. The algorithm is very simple and does not involve any communication between processes: each dedicated core computes an estimation of the computation time of an iteration from a first run of the simulation (about 230 sec on Kraken). This time is then divided into as many slots as dedicated cores. Each dedicated core then waits for its slot before writing. This avoids access contention at the level of the file system. Evaluating this strategy on 2304 cores on Kraken, the aggregate throughput achieves 13.1 GB/s on average, instead of 9.7 GB/s when this algorithm is not used.

These strategies have also been evaluated on 912 cores of Grid'5000. All these results are presented in Figure 16 in terms of write time in the dedicated cores. As we can see, the scheduling strategy allows to decrease the write time in both platforms. Compression however introduces an overhead on Kraken, thus we are facing a tradeoff between reducing the required storage space and the spare time. A potential optimization would be to enable or disable compression at run time depending on the need for sparing time or storage space.

6 Discussion and related work

In this section, we evaluate the benefits of our approach by computing mathematical bounds of effectiveness, and by comparing to experimental ascertainments. We then position our approach with respect to other related research.

6.1 Are all my cores really needed?

Let us call W_{std} the time spent writing and C_{std} the computation time between two I/O phases with a standard approach, C_{ded} the computation time when the same workload is divided across one less core in each node. We here assume that the I/O time is completely hidden or at least negligible when using the dedicated core (which is experimentally verified) from the point of view of the simulation, and we call W_{ded} the time that the dedicated core spends writing. A theoretical performance benefit of our approach then occurs when

$$W_{std} + C_{std} > max(C_{ded}, W_{ded})$$

Assuming an optimal parallelization of the program across N cores per node, and the worst case for Damaris where $W_{ded} = N * W_{std}$, we show that this inequality is true when the program spends at least $p\%$ in I/O phase, with $p = \frac{100}{N-1}$. As an example, with 24 cores $p = 4.35\%$, which is already under the 5% usually admitted for the I/O phase of such applications. Thus assuming that the application effectively spends 5% of the time writing data, on a machine featuring more than 24 cores per node, it is more efficient to dedicate one core per node to hide the I/O phase.

However this is a simplified theoretical estimation. In practice, most HPC simulations doesn't exhibit a linear scalability. We have run CM1 with the exact same configuration and network topology on Grid'5000, dividing the workload across 24 cores per node first, then 23 cores per node. 200 iterations with 24 cores per node where done in 44'17" while only 41'46" were necessary to run 200 iterations with 23 cores per node. Similar ascertainment was made on Kraken, especially as we increase the total number of cores, using 11 cores per nodes instead of 12 already produces an improvement. The final statistics output by CM1 showed that all the phases that use all-to-all communications benefit from leaving one core out, thus actually reducing the computation time as memory contention is reduced. Such a behavior of multicore architectures is explained in [19] and motivates future studies regarding the appropriate number of cores that can be dedicated to services as Damaris does. Moreover, writing more data from a reduced number of cores has shown to be much more efficient than the worst case used in the theoretical bound.

6.2 Positioning Damaris in the "I/O landscape"

Through its capability of gathering data into larger buffers and files, Damaris can be compared to the ROMIO data aggregation feature [31]. Yet, data aggregation is performed synchronously in ROMIO, so all cores that do not perform actual writes in the file system must wait for the aggregator processes to complete their operations. Through space-partitioning, Damaris can perform data aggregation and potential transformations in an asynchronous manner and still use the idle time remaining in the dedicated cores.

Other efforts are focused on overlapping computation with I/O in order to reduce the impact of I/O latency on overall performance. Overlap techniques can be implemented directly within simulations [25], using asynchronous communications. Yet non-blocking file operations primitives are not part of the current MPI-2 standard. A quantification of the potential benefit of overlapping communication and computation is provided in [27], which demonstrates methodologies that can be extended to communications used by I/O. Our Damaris approach exploits the potential benefit of this overlap.

Other approaches leverage data-staging and caching mechanisms [24, 12], along with forwarding approaches [1] to achieve better I/O performance. Forwarding routines usually run on top of dedicated resources in the platform, which are not configurable by the end-user. Moreover, these dedicated resources are shared by all users, which leads to multi-application access contention and thus to jitter. However, the trend towards I/O delegate systems underscores the need for new I/O approaches. Our approach follows this idea but relies on dedicated I/O cores at the application level rather than hardware I/O-dedicated or forwarding nodes, with the advantage of letting the user configure its dedicated resources to best fit its needs.

The use of local memory to alleviate the load on the file system is not new. The Scalable Checkpoint/Restart by Moody et al. [18] already makes use of node-local storages to avoid the heavy load caused by periodic global checkpoints. Their work yet does not use dedicated resources or threads to handle or process data, and the checkpoints are not asynchronous. But it exemplified the fact that node-local memory can be used to delay or avoid file system's interactions. When additional SSDs, Flash memory or disks are attached to compute nodes, both SCR and Damaris can leverage it to avoid consuming to much of RAM.

Some research efforts have focused on reserving computational resources as a bridge between the simulation and the file system or other back-ends such as visualization engines. In such approaches, I/O at the simulation level is replaced by asynchronous communications with a middleware running on a separate set of computation nodes, where data is stored in local memory and processed prior to effective storage. PreDatA [33] is such an example: it performs in-transit data manipulation on a subset of compute nodes prior to storage, allowing more efficient I/O in the simulation and more simple data analytics, at the price of reserving dedicated computational resources. The communication between simulation nodes and PreDatA nodes is done through the DART [10] RDMA-based transport method, hidden behind the ADIOS interface which allows the system to any simulation that has an ADIOS I/O backend. However, given the high ratio between the number of nodes used by the simulation and the number of nodes used for data processing, the PreDatA middleware is forced to perform streaming data processing, while our approach using dedicated cores in the simulation nodes allows us to keep the data longer in memory or any node-local storage devices, and to smartly schedule all data operations and movements. Clearly some simulations would benefit from one approach or the other, depending on their needs in terms of memory, I/O throughput and computation performance, but also the two approaches – dedicating cores or dedicating nodes – are complementary and we could imagine a framework that make use of the two ideas.

Closest to our work are the approaches described in [14] and [17]. While the general goals of these approaches are similar (leveraging service-dedicated cores for non-computational tasks), their design is different, and so is the focus and the (much lower) scale of their evaluation. [14] is an effort parallel to ours. It mainly explores the idea of using dedicated cores in conjunction with the use of SSDs to improve the overall I/O throughput. Architecturally, it relies on a FUSE interface, which introduces useless copies and reduces the degree of coupling between cores. In [17], active buffers are handled by dedicated processes that can run on any node and interact with cores running the simulation through network. In contrast to both approaches, Damaris makes a much more efficient design choice using the shared intra-node memory, thereby avoiding costly copies and buffering. The approach defended by [14] is demonstrated on a small 32-

node cluster (160 cores), where the maximum scale used in [17] is 512 cores on a POWER3 machine, for which the overall improvement achieved for the global run time is marginal. Our experimental analysis is much more extensive and more relevant for today's scales of HPC supercomputers: we demonstrate the excellent scalability of Damaris on a real supercomputer (Kraken, ranked 11th in the Top500 supercomputer list) up to almost 10,000 cores, with the CM1 tornado simulation, one of the target applications of the Blue Waters postpetascale supercomputer project. We demonstrate not only a speedup in I/O throughput by a factor of 15 (never achieved by previous approaches), but we also demonstrate that Damaris totally hides jitter and substantially cuts down the application run time at such high scales (see Figure 13): execution time is cut by 35% compared to the file-per-process approach with 9,216 cores, wheras the largest experiment in [17] (512 cores) with a real-life application only shows a very marginal improvement in execution time. With Damaris, the execution time for CM1 at this scale is even divided by 3.5 compared to approaches based on collective-I/O! Moreover, we further explore how to leverage the spare time of the dedicated cores (e.g. we demonstrate that it can be used to compress data by a factor of 6).

7 Conclusions

Efficient management of I/O variability (*aka* jitter) on today's Petascale and Post-Petascale HPC infrastructures is a key challenge, as jitter has a huge impact on the ability to sustain high performance at the scale of such machines (hundreds of cores). Understanding its behavior and proposing efficient solutions to reduce its effects is critical for preparing the advent of Exascale machines and their efficient usage by applications at the full machine scale. This is precisely the challenge addressed by this paper. Given the limited scalability of existing approaches to I/O in terms of I/O throughput and also given their high I/O performance variability, we propose a new approach (called Damaris) which originally leverages dedicated I/O cores on multicore SMP nodes in conjunction with an efficient usage of shared intra-node memory. This solution provides the capability not only to better schedule data movement through asynchronous I/O, but also to leverage the dedicated I/O cores to do extra useful data preprocessing prior to storage or visualization (such as compression or formatting).

Results obtained with one of the challenging target applications of the Blue Waters postpetascale supercomputer (now being delivered at NCSA), clearly demonstrate the benefits of Damaris in experiments with up to 9216 cores performed on the Kraken supercomputer (ranked 11th in the Top500 list). Damaris completely hides I/O jitter and all I/O-related costs and achieves a throughput 15 times higher than standard existing approaches. Besides, it reduces application execution time by 35% compared to the conventional file-per-process approach. Execution time is divided by 3.5 compared to approaches based on collective-I/O! Moreover, it substantially reduces storage requirements, as the dedicated I/O cores enable overhead-free data compression with up to 600% compression ratio. To the best of our knowledge, no concurrent approach demonstrated such improvements in all these directions at such scales. The high *practical* impact of this promising approach has recently been recognized by application communities expected to benefit from the Blue Waters supercomputer and, for these reasons, Damaris was formally validated to be used by these applications on Blue Waters.

Our future work will focus on several directions. We plan to quantify the optimal ratio between I/O cores and computation cores within a node for several classes of HPC simulations. We are also investigating other ways to leverage spare time of I/O cores. A very promising direction is to attempt a tight coupling between running simulations and visualization engines, enabling direct access to data by visualization engines (through the I/O cores) while the simulation is running. This could enable efficient inline visualization without blocking the simulation, thereby reducing the pressure on the file systems. Finally, we plan to explore coordination strategies of I/O cores in order to implement distributed I/O scheduling.

Acknowledgments

This work was done in the framework of a collaboration between the KerData INRIA - ENS Cachan/Brittany team (Rennes, France) and the NCSA (Urbana-Champaign, USA) within the Joint INRIA-UIUC Laboratory for Petascale Computing. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `http://www.grid5000.fr/` for details) and on Kraken at NICS (see `http://www.nics.tennessee.edu/`). The authors also acknowledge the PVFS developers, the HDF5 group, Kraken administrators and the Grid'5000 users, who helped properly configuring the tools and platforms used for this work. We acknowledge Robert Wilhelmson (NCSA, UIUC) for his insights on the CM1 application and Dave Semeraro (NCSA, UIUC) for our fruitful discussions on visualization/simulation coupling.

References

[1] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, September 2009.

[2] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481, 2006.

[3] G. H. Bryan and J. M. Fritsch. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review*, 130(12):2917–2928, 2002.

[4] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. *International Parallel and Distributed Processing Symposium*, pages 1–11, 2009.

[5] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, Berkeley, CA, USA, 2000. USENIX Association.

[6] C. Chilan, M. Yang, A. Cheng, and L. Arber. Parallel I/O performance study with HDF5, a scientific data package, 2006.

[7] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Non-contiguous I/O through PVFS. page 405, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[8] P. Dickens and J. Logan. Towards a high performance implementation of MPI-I/O on the Lustre file system. *On the Move to Meaningful Internet Systems OTM 2008*, 2008.

[9] P. Dickens and R. Thakur. Evaluation of Collective I/O Implementations on Parallel Architectures. *Journal of Parallel and Distributed Computing*, 61(8):1052 – 1076, 2001.

[10] C. Docan, M. Parashar, and S. Klasky. Enabling high-speed asynchronous data extraction and transfer using DART. *Concurrency and Computation: Practice and Experience*, pages 1181–1204, 2010.

[11] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan. Lustre: Building a file system for 1000-node clusters, 2003.

[12] F. Isaila, J. G. Blas, J. Carretero, R. Latham, and R. Ross. Design and evaluation of multiple level data staging for Blue Gene systems. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[13] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. page 39. IEEE, 2006.

[14] M. Li, S. Vazhkudai, A. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[15] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the IO performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[16] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[17] X. Ma, J. Lee, and M. Winslett. High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Transactions on Parallel and Distributed Systems*, 17:193–204, 2006.

[18] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. *SC Conference*, pages 1–11, 2010.

[19] S. Moore. Multicore is bad news for supercomputers. *Spectrum, IEEE*, 45(11):15, 2008.

[20] Mpich2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[21] NCSA. BlueWaters project, `http://www.ncsa.illinois.edu/BlueWaters/`.

[22] NICS. Kraken Cray XT5, `http://www.nics.tennessee.edu/computing-resources/kraken`.

[23] NICS. `http://www.nics.tennessee.edu/io-tips`.

[24] A. Nisar, W. keng Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008.

[25] C. M. Patrick, S. Son, and M. Kandemir. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO. *SIGOPS Oper. Syst. Rev.*, 42:43–49, October 2008.

[26] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS an Optimized Implementation of MPI-IO on Top of GPFS. page 58, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[27] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, page 17, 2006.

[28] F. Schmuck and R. Haskin. GPFS A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*. Citeseer, 2002.

[29] H. Shan and J. Shalf. Using IOR to Analyze the I/O performance for HPC Platforms. In *Cray User Group Conference 2007*, Seattle, WA, USA, 2007.

[30] D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. In *IEEE Workload Characterization Symposium*, pages 137–149. IEEE Computer Society, 2005.

[31] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. *Symposium on the Frontiers of Massively Parallel Processing*, page 182, 1999.

[32] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. Parallel I/O performance: From events to ensembles. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010.

[33] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA – preparatory data analytics on peta-scale machines. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399